Learning PySpark: Removing Leading Zeros from DataFrame Columns

Authored by

Mohammed looti

November 11, 2025

Data cleansing is a fundamental step in any robust data pipeline, especially when dealing with legacy systems or disparate data sources. A common challenge encountered when processing identifiers or numerical codes within an PySpark DataFrame is the presence of leading zeros. While these zeros might be necessary for fixed-width data formats, they often obscure the true numerical value and complicate downstream analysis or joins. In this comprehensive guide, we demonstrate the most efficient and standard method in PySpark for eliminating these redundant characters.

Utilizing PySpark's regexp_replace Function for Precision

The preferred tool for complex string manipulation in PySpark is the **functions.regexp_replace** function. This powerful function leverages the flexibility of Regular expressions (regex) to identify patterns, such as sequences of leading zeros, and replace them with a specified replacement string. This approach ensures that only the initial zero characters are targeted, leaving internal or trailing zeros untouched, thus preserving the integrity of the remaining data.

The general syntax required to implement this transformation is straightforward and highly effective for data normalization tasks. You can use the following standard pattern to remove leading zeros from a string column in a PySpark DataFrame:

from pyspark.sql import functions as F

# Define the transformation to remove leading zeros from the 'employee_ID' column
df_new = df.withColumn('employee_ID', F.regexp_replace('employee_ID', r'^*', ''))

In this snippet, we are applying the replacement logic to the designated column, here named `employee_ID`. Crucially, the pattern `r'^*'` is designed to capture the entire sequence of zeros that initiate the string, replacing that entire sequence with an empty string (''). This specific application ensures that only the zeros preceding the first non-zero digit are removed, while any zeros embedded later in the identifier remain untouched.

Dissecting the Regular Expression Pattern: `r'^*'`

Understanding the Regular expression (regex) pattern `r'^*'` is essential to guaranteeing the accuracy of the data transformation. This simple yet critical pattern is the core mechanism that distinguishes true leading zeros from legitimate internal zeros. The use of the `r` prefix denotes a raw string, which is standard practice in Python when dealing with regex patterns to avoid issues with backslash escaping.

Let us break down the components of this powerful pattern:

`^` **(Caret Anchor):** This character signifies the start of the string. By anchoring the pattern to the beginning, we ensure that the match only occurs at the very start of the column value. This is what defines the "leading" aspect of the zeros.

(Character Set): This specifies that we are looking for the literal digit zero.

`*` **(Quantifier):** The asterisk is a quantifier meaning "zero or more occurrences" of the preceding character (in this case, '0'). Combining `*` means we will match any continuous sequence of the digit zero.

Therefore, `r'^*'` matches any sequence of zero or more zeros that appear at the **start** of the string. When this matched sequence is replaced by an empty string (`''`), the leading zeros effectively vanish, leaving behind the true numerical component of the identifier.

Practical Demonstration: Removing Leading Zeros from a Column in PySpark

To illustrate this process, let us work with a sample DataFrame that simulates employee identification data. This data set contains the `employee_ID` column, which, due to formatting constraints or legacy input methods, includes pervasive leading zeros that require normalization before further analytical processing can occur.

Setting Up the Sample PySpark DataFrame

We begin by initiating a **SparkSession**, the entry point for all PySpark functionality, and defining our sample data structure. This structure includes employee identifiers where leading zeros vary in count, providing a comprehensive test case for our regex solution. The initial data creation process is critical for verifying the effectiveness of the subsequent transformation steps.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define the raw data structure:
data = ,
,
,
,
,
,
]

```
# Define the necessary column headers
columns =

# Create the PySpark DataFrame
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure, highlighting the leading zeros
df.show()
```

```
+-----------+-----+
|employee_ID|sales|
+-----------+-----+
| 000501| 18|
| 000034| 33|
| 009230| 12|
| 000451| 15|
| 000239| 19|
| 002295| 24|
| 011543| 28|
+-----------+-----+
```

As evident from the output, the `employee_ID` column currently stores identifiers as strings padded with varying numbers of leading zeros. Our objective is to normalize these values by stripping the initial zero padding while ensuring that the data type remains suitable for the resulting numerical identifier.

Applying the Transformation and Reviewing Results

With the DataFrame initialized, we can now apply the previously defined logic using `F.regexp_replace`. This operation creates a new column (or overwrites the existing one, as shown here) with the cleaned strings. It is critical to import the `functions` module from `pyspark.sql`, typically aliased as `F`, to access the necessary string manipulation tools.

from pyspark.sql import functions as F

```
# Apply the regex pattern to remove any leading zeros
df_new = df.withColumn('employee_ID', F.regexp_replace('employee_ID', r'^*', ''))

# View the updated DataFrame
df_new.show()
```

```
+-----------+-----+
|employee_ID|sales|
+-----------+-----+
| 501| 18|
| 34| 33|
| 9230| 12|
| 451| 15|
| 239| 19|
| 2295| 24|
| 11543| 28|
+-----------+-----+
```

The resulting DataFrame, `df_new`, clearly demonstrates that the leading zeros have been successfully eliminated from every entry in the `employee_ID` column. For instance, '000501' is transformed into '501', and '009230' correctly becomes '9230', demonstrating that the internal zero was preserved, fulfilling the core requirement of the transformation. Note that we used the PySpark **regexp_replace** function to replace the leading zeros in each string with nothing.

Alternative Methods: Using PySpark's Built-in String Functions

While `regexp_replace` is the most versatile method, especially when dealing with complex patterns, PySpark also offers simpler string functions that can achieve similar results, though often with less control or requiring an additional type casting step. If the column contains strictly numerical data stored as strings, casting the column directly to a numerical type will automatically discard the leading zeros.

The primary alternative methods for achieving zero removal include:

Casting to Integer or Long: If the resulting number fits within standard integer limits, casting the string column to `IntegerType` or `LongType` will inherently drop all leading zeros. This is often the fastest method when the goal is a numeric output. If the final output must remain a string, the casting method requires two steps: cast String to Integer, and then cast the resulting Integer back to String. This ensures the zero removal is successful without the need for complex regex, but it risks data loss if the string contains non-numeric characters.

Using `ltrim`: While `ltrim` is typically used for removing whitespace, it can be configured to remove specific padding characters from the left side of a string. However, `ltrim` is less precise than regex because it will continue trimming the specified character until it hits a different character, which may not always be what is intended if the data is highly varied.

The choice between `regexp_replace` and casting depends heavily on the data integrity requirements and whether the column is guaranteed to be purely numeric. For maximum safety and pattern matching precision on identifiers that might contain mixed characters, the regexp_replace function using the `r'^*'` pattern remains the highly recommended best practice in enterprise data environments.

Additional Resources for Advanced PySpark Techniques

Mastering string manipulation is crucial for effective data engineering in PySpark. The `pyspark.sql.functions` module provides a rich array of tools beyond `regexp_replace` that handle various data transformation needs.

The following tutorials explain how to perform other common tasks in PySpark:

Detailed documentation for Apache Spark and its Python API.

Guides on using other PySpark string functions like `trim`, `substring`, and `translate`.

Tutorials covering data type optimization and null value handling in large-scale DataFrames.